RICE UNIVERSITY

Accelerated High-Performance Compressive Sensing using the Graphics Processing Unit

by

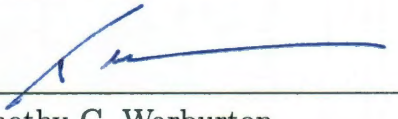
Nabor Reyna Jr.

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Arts

Approved, Thesis Committee:

Wotao Yin, Chair
Assistant Professor of Computational and Applied Mathematics



Timothy C. Warburton
Associate Professor of Computational and Applied Mathematics

Yin Zhang
Professor of Computational and Applied Mathematics

Houston, Texas

April, 2011

ABSTRACT

Accelerated High-Performance Compressive
Sensing using the Graphics Processing Unit

by

Nabor Reyna Jr.

This thesis demonstrates the advantages of new practical implementations of compressive sensing (CS) algorithms tailored for the graphics processing unit (GPU) using a software platform called Jacket. There exist many applications which utilize CS including medical imaging, signal processing and data acquisition which have benefited from advancements in CS. However, as problems become larger not only do they become more difficult to solve but also more computationally expensive. In light of this, existing CS algorithms are augmented for practical use on the GPU, reaping performance gains from the highly parallel architecture of the GPU. I discuss the issues associated with this transition and analyze the effects of such a movement, as well as provide results exhibiting advantages of using GPU-based methods.

Acknowledgments

I am indebted to many in my academic career. First, I would like thank my thank my advisor and mentor, Dr. Wotao Yin whose skills as a mathematician have inspired me greatly. I also thank Dr. Timothy C. Warburton and Dr. Yin Zhang, for their thoughtful contributions and revisions.

I would like to acknowledge my fellow members of the Sparse Optimization Group, supported by a VIGRE grant from the National Science Foundation: Jorge Castanon, Wolfgang Stefan, Chengbo Li and Nancy Okeudo. Some sections of this thesis became clearer after insight gained during our seminar sessions.

I am especially thankful to Russell Carden, for sitting behind me and keeping me on task. Russell also helped me prepare for my defense and had many insightful revisions of this work. I would like to thank his family for the many smiles they have put on my face over the years, especially when pastries were involved.

I would like to acknowledge Dr. Maria Cristina Villalobos for all of the great advice she has given me over the years. I thank Dr. Tapia for allowing me to become a part of the AGEP community and a part of his extended family.

I want to thank Josef Sifuentes not only for his refined palette but also for his willingness to always help.

To Theresa Chatman who has always left her door open for me to come vent. I am deeply indebted to her. To all of my fellow classmates (CAAM 2008). To all that I have left out and know they should be thanked. Thank you!

Finally, I am extremely grateful to have the love and unwavering support of my family. I thank my parents and grandparents who have always encouraged me to work hard and strive for the best.

Contents

Illustrations

Chapter 1

Introduction

This thesis demonstrates a new practical implementation of compressive sensing (CS) algorithms tailored for the graphics processing unit (GPU) using a software package called Jacket. In recent years there has been a surge of attention placed on both compressive sensing and general-purpose computing on the graphical processing unit (GPGPU) ([4], [32]). This attention can be largely credited to the vast multitude of applications in which both play key roles. For example, CS has been able to speed up the process of reconstructing pediatric MRI images by a factor of seven [41] and the GPU's computational performance has been able to shorten the time it takes for a simulation to run [20]. Capitalizing on the many advancements of both CS solvers and GPU performance, this work aims at creating methods which will provide even faster results.

1.1 Compressive Sensing

The field of compressive sensing, which some credit Santosa and Symes [37] as being the pioneers, has changed how signals are acquired, stored, transmitted and processed. In the early 2000s work by Donoho [40] and Candes, Romberg and Tao [8] has brought much attention to the field.

Compressive sensing allows for a more efficient way to sample and reconstruct signals compared to traditional data acquisition and reconstruction techniques which may under use samples. Researchers have applied this new CS sampling modality in many settings, such as face recognition [43], medical imaging [41] and statistics [38] to name a few.

Despite the extensive amount of applications, there does not exist much literature on the implementation of CS methods for the GPU. In this thesis I take advantage of the computational gain that the GPU provides to create methods which provide a time saving; comparisons to their central processing unit (CPU) counterparts are presented.

1.2 Graphics Processing Unit

Graphics processing units are highly parallel, multi–thread, many core processors, which have also shared similar popularity to compressive sensing due in large part to the large peak performances that the cards provide. Researchers have observed that computations done in parallel provide a route for completing computationally intensive numerical tasks faster.

Initially developed for graphical rendering, GPUs have come a long way and are now known as computational workhorses because of the peak computational performance and high memory bandwidth [10]. Furthermore, GPUs are now used in an array of applications such as a molecular dynamics [39], computational finance [20], medical imaging [31] and many other applications. These applications take advan-

tage of state-of-the-art graphics hardware and demonstrate the capability of boosting performance by several orders of magnitude.

However, it should be noted that in general, efficient implementations of numerical algorithms on the GPU are difficult to develop due to the complexity of the architectures and their technical specifications [5]. In order to overcome this hurdle, in this thesis the use of a software platform called Jacket created by AccelerEyes aids in creating implementations [1].

This platform allows implementations of methods to be composed with traditional MATLAB language along with a few new classes used to access the computing and visual capability of the GPU. My research helps make CS techniques more appealing, as this work concentrates on reaping faster computational times in solving compressive sensing problems.

1.3 Overview

The remainder of the thesis develops the key principles that will be needed to understand the work being presented. In Chapter 2 a more extensive background of compressive sensing is given along with some numerical methods. Chapter 3 presents NVIDIA's graphics processing unit and its uses for scientific computation. I then provide a recipe which uses the theory of CS and GPUs along with Jacket to create my methods in Chapter 4. Numerical simulations and results are discussed in Chapter 5. In the last chapter I present my concluding remarks followed by possible directions in which this work may be taken.

Chapter 2

Compressive Sensing

As technology becomes more prevalent in our society, an ever increasing amount of data becomes available. The challenge that arises is how to best handle such large amounts of data. The Nyquist sampling theorem gives direction towards reconstructing a signal dependably, i.e. without loss of information, by stating that a signal must be sampled at a rate of at least two times that of the highest frequency in the signal [4].

This work concentrates on the reconstruction of signals using a small number of data samples, especially for applications related to medical imaging systems. In medical imaging systems increasing the sampling rate can be very expensive if not impossible when using traditional signal acquisition. The way traditional sampling works is that sampling of the full signal takes place followed by computing a complete set of transform coefficients [6]. The signal is then encoded using only the largest coefficients, while discarding all the others. It is clear to see that this process of massive data acquisition followed by compression can be extremely wasteful.

With the increase in sample rates several questions become important such as, why acquire so much data when most will be disregarded in the compression? [12] Or can important samples be taken directly? Romberg [36] investigates further by asking "Is there a way we can build the data compression directly into the acquisition?"

Fortunately, this is where the theory of compressive sensing comes in, with further details given in this chapter.

2.1 What is compressive sensing?

Compressive Sensing is a simple and efficient signal acquisition protocol that provides a way to take samples, independent of the signal type, at a lower rate than the Nyquist sampling rate. The small number of samples are then used to reconstruct compressible sparse signals with the use of computational power to solve a non-smooth convex optimization problem from what appears to be an incomplete set of measurements [7].

According to Donoho [12], the theory of compressive sensing (sometimes referred to as compressive sampling) reappeared in 2004, when significant results for the minimum number of samples needed to reconstruct a signal were discovered. Nevertheless this relatively new field contains an immense amount of literature. These may be due to the several advantages of compressive sensing; for example, the need for fewer samples to reconstruct a signal allows CS to sample signals faster. As a consequence of using fewer samples, CS allows for acquisition systems to have lower energy consumption, by reducing memory and sensor requirements. Working with higher dimensional data also becomes more feasible considering that the number of samples to reconstruct the signals will not require the number of samples to grow very large.

These simple advantages of CS make it appealing to a broad range of real world scenarios such as medical imaging [48], statistics [38], face recognition [43] and wireless

networks [29]. For this reason, I propose new methods that can create the reconstructions using compressive sensing with the expectation to save time. This work will in turn have a positive impact in many fields. Further literature containing more theoretical implications of CS exist and the curious reader should consult other papers (i.e. [6, 4]) for a deeper understanding of compressive sensing.

2.2 Compressive sensing on the GPU

Even with many resources existing for CS, covering an array of topics from theory to practice, there only exist a handful of papers which try to utilize the graphics hardware or parallel architectures to enhance the performance of CS methods. Lee and Wright [24] demonstrate an extension of work previously done with the sparse reconstruction by separable approximation algorithm (SpaRSA) through an implementation on a GPU card. These authors were able to report results with an average speedup of about 34 times based on their comparisons between two different implementations of SpaRSA for the CPU and the GPU. It is important to note that the SpaRSA algorithm used in Lee and Wright's study is not one of the fastest for the CPU [44].

Borghi et al. [5] has also reported speedup through implementations on graphics hardware. However, their results also show that multicore CPUs can offer comparable performances with GPUs when their parallel features are used efficiently. A good analysis is also provided in Borghi et al. on the implications found for implementing compressive sensing methods on different architectures.

It is well known that supercomputers provide faster solutions, thus the computational performance of the GPU has influenced me to seek implementations with faster recovery times. Similar to Lee and Wright as well as Borghi et al., I present two new methods for the GPU, based on an extension of the work in [48] and [50].

2.3 Existing compressive sensing solvers

In this thesis, I concentrate on two existing methods that are able to retrieve fast reconstructions. In particular, I use the reconstruction from partial Fourier data (RecPF) and reconstruction using Toeplitz and circulant matrices (RecPC) methods, both of which have proven to be some of the faster methods to date.

2.3.1 Reconstruction from Partial Fourier Data (RecPF)

RecPF, as defined in [48], reconstructs signals by minimizing the sum of three terms each of which corresponds to: total variation, ℓ_1-norm regularization and least squares data fitting, while using partial Fourier data to reconstruct signals. This model was investigated in [15, 18, 28, 27] and was reported to reconstruct high quality MR images from a small number of Fourier coefficients [28]. Specifically the method minimizes x that solves the following:

$$\min_x \ \alpha \mathrm{TV}(x) + \beta \|\Psi x\|_1 + \frac{\mu}{2} \|P\mathcal{F}(x) - f_p\|_2^2,$$

where x is the signal/image that is to be reconstructed and $\mathrm{TV}(x)$ represents the total variation term. The second term renders the ℓ_1-norm regularization, where Ψ is a sparsifying basis; i.e., the signal becomes sparse when represented in this basis.

The last term is sometimes referred to as the fidelity term, which is a least squared data fitting term. A partial Fourier matrix, denoted $\mathcal{F}_p$, is created by taking a selection matrix which has p randomly select rows from a $n \times n$ identity matrix then multiplying with the full Fourier discretization $\mathcal{F}$, giving p randomly select rows of $\mathcal{F}$. Also, f_p is a vector of partial Fourier coefficients belonging to the desired signal in the reconstruction. RecPF can be easily applied to reconstructing magnetic resonance images (MRI), which are discussed in this thesis.

An alternating minimization scheme is used for the minimization of RecPF, where the main computation involve shrinkage and fast Fourier transforms (FFTs). RecPF may also use discrete cosine transforms (DCTs) when available data is in the DCT domain. However, in this thesis signals in the DCT domain are not studied as the focus is on MRI signal reconstruction.

This method allows for great performance in reconstructing signals but has a drawback for practical implementations because random Fourier matrices are often difficult and costly to implement in hardware realizations. Random Toeplitz and circulant matrices can be easily (or even naturally) realized in various applications. The following section introduces a fast algorithm for reconstructing signals from incomplete Toeplitz and circulant measurements.

2.3.2 Reconstruction using Toeplitz and circulant matrices (RecPC)

The use of Toeplitz and circulant matrices provides a route for making hardware realizations feasible. A Toeplitz matrix is one where each descending diagonal from left to right has a constant value. For example, the following matrix is an $n \times n$

Toeplitz matrix:

$$T = \begin{bmatrix} t_n & t_{n-1} & \cdots & \cdots & t_1 \\ t_{n+1} & t_n & t_{n-1} & \cdots & t_2 \\ \ddots & \ddots & \ddots & \ddots & \vdots \\ t_{2n-2} & \ddots & \ddots & t_n & t_{n-1} \\ t_{2n-1} & t_{2n-2} & \cdots & t_{n+1} & t_n \end{bmatrix}.$$

An example of a circulant matrix is given below. A circulant matrix is a special kind of Toeplitz matrix where each row has been rotated one element to the right relative to the preceding row:

$$C = \begin{bmatrix} c_1 & c_2 & \cdots & \cdots & c_n \\ c_n & c_1 & c_2 & \ddots & c_{n-1} \\ \ddots & \ddots & \ddots & \ddots & \vdots \\ c_3 & \ddots & \ddots & c_1 & c_2 \\ c_2 & c_3 & \cdots & c_n & c_1 \end{bmatrix}.$$

The quest for a feasible hardware realization is what has led to the creation of reconstruction using Toeplitz and circulant matrices (RecPC) method [50]. Thus the problem becomes finding the best x such that,

$$\min_x \ \alpha \mathrm{TV}(x) + \beta \|\Psi x\|_1 + \frac{\mu}{2} \|PCx - b\|_2^2.$$

Again, x is the signal/image to be reconstructed and $\mathrm{TV}(x)$ is the total variation term. The Ψ term still represents the sparsifying basis to be used but now the selection operator P is presented. This selection operator is an identity matrix of the same order as the signal desired but has p rows. The major difference for this

formulation is C, which is a block-circulant matrix as opposed to the Fourier matrix that was used before. For two-dimensional signals $\widetilde{C}$ replaces C, which is similarly defined as above, with the exception that the entries are now matrices instead of scalar values.

$$\widetilde{C} = \begin{bmatrix} C_1 & C_2 & \cdots & \cdots & C_n \\ C_n & C_1 & C_2 & \ddots & C_{n-1} \\ \ddots & \ddots & \ddots & \ddots & \vdots \\ C_3 & \ddots & \ddots & C_1 & C_2 \\ C_2 & C_3 & \cdots & C_n & C_1 \end{bmatrix}$$

My interest in RecPC lie on the proposed use of Toeplitz and circulant matrices for compressive MR imaging by Liang et al. [26]. Also, the results given by Yin et al. [50] suggest that if this measurement scheme becomes feasible for MRI systems, then potentially the number of measurements could be substantially smaller than the number that is required by Fourier schemes in a CS setting.

2.4 Solving RecPF and RecPC

In this section a brief but comprehensive approach is taken to show how to attain solutions for both the RecPF and RecPC methods, by using the augmented Lagrangian method (ALM) and alternating direction method (ADM). A variable-splitting technique is applied to a problem in the form of $\min\{f(Lx) + g(x)\}$, which equivalently obtains $\min\{f(y) + g(x) : Lx - y = 0\}$. This technique allows for the problem's augmented Lagrangian $f(y) + \langle \lambda, Lx - y \rangle + \|Lx - y\|_2^2 + g(x)$ to be used.

The formulation for the RecPC model, given by

$$\min_x \ \alpha \mathrm{TV}(x) + \beta \|\Psi x\|_1 + \frac{\mu}{2}\|PCx - b\|_2^2$$

is used to describe this approach. Also, the main difficulty in solving this model is caused by the non-differentiability of its first and second terms.

For this section the standard treatment of two-dimensional images (or higher dimensional data) will be to vectorize the signal into one-dimensional vectors [48]. At the end of this subsection the subtle differences between RecPC and RecPF will be mentioned, as well as a template for solving the RecPF model.

For convenience of notation, I will let $\psi_i^\top$ be the ith row of Ψ, and write the discretized total variation as $\mathrm{TV}(x) = \sum_{i=1}^{n^2} \|D_i x\|$. Using this notation, the RecPC model becomes

$$\min \alpha \sum_{i=1}^{n^2} \|D_i x\| + \beta \sum_{i=1}^{n^2} |\psi_i^\top x| + \frac{\mu}{2}\|PCx - b\|_2^2, \tag{2.1}$$

for a two-dimensional image. Then by introducing $\mathbf{y} = [\mathbf{y}_1, \ldots, \mathbf{y}_{n^2}]$, where each $\mathbf{y}_i \in \mathbb{R}^2$, and $z, u \in \mathbb{R}^{n^2}$, problem (2.1) is transformed to

$$\begin{aligned} \min_{\mathbf{y},z,u,x} \Big\{ \alpha \sum_i \|\mathbf{y}_i\| + \beta \sum_i |z_i| + \frac{\mu}{2}\|Pu - b\|_2^2 \ ; \\ u = Cx, \ \mathbf{y}_i = D_i x, \ z_i = \psi_i^\top x, \ i = 1, 2, \ldots, n^2 \Big\} . \end{aligned} \tag{2.2}$$

The objective function of (2.2) now has three independent terms, which are only connected through the linear constraints. A quadratic penalty technique is used to relax the equality constraints and penalize their violations, which dates back to the

work done by Courant [11]. I define the quadratic functions as

$$\begin{aligned} \varphi_1(s,t,\rho) &= |s| - \rho(s-t) + \tfrac{\beta_1}{2}\|s-t\|^2 \\ \varphi_2(\mathbf{s},\mathbf{t},\nu) &= \|\mathbf{s}\| - \nu^\top(\mathbf{s}-\mathbf{t}) + \tfrac{\beta_2}{2}\|\mathbf{s}-\mathbf{t}\|^2, \end{aligned} \tag{2.3}$$

where $s,t,\rho \in \mathbb{R}$, $\mathbf{s},\mathbf{t},\nu \in \mathbb{R}^2$ and $\beta_1,\beta_2 > 0$ are parameters (not to be confused with β). Then, the augmented Lagrangian function of (2.2) can be written as

$$\begin{aligned} \mathcal{L}_{\mathcal{A}}(\mathbf{y},z,u,x,\lambda) = \; & \alpha\sum\nolimits_i \varphi_2\left(\mathbf{y}_i, D_i x, (\lambda_2)_i\right) + \beta\sum\nolimits_i \varphi_1\left(z_i, \psi_i^\top x, (\lambda_1)_i\right) \\ & + \frac{\beta_3}{2}\left\| u - Cx - \frac{\lambda_3}{\beta_3}\right\|^2 + \frac{\mu}{2}\|Pu-b\|^2, \end{aligned} \tag{2.4}$$

where $\beta_3 > 0$ is a penalty parameter and $\lambda = (\lambda_1,\lambda_2,\lambda_3)$ contains the Lagrangian multipliers. For each i, $(\lambda_1)_i \in \mathbb{R}$ and $(\lambda_2)_i \in \mathbb{R}^2$, and $\lambda_3 \in \mathbb{R}^{n^2}$. Given $(\mathbf{y}^k, z^k, u^k, x^k)$ and λ^k, the classical augmented Lagrangian method [19, 35](ALM) for (2.2) iterates as

$$\begin{cases} (\mathbf{y}^{k+1}, z^{k+1}, u^{k+1}, x^{k+1}) \leftarrow \arg\min \mathcal{L}_{\mathcal{A}}\left(\mathbf{y}, z, u, x, \lambda^k\right), \\ (\lambda_1)_i^{k+1} \leftarrow (\lambda_1)_i^k - \gamma_1\beta_1(z_i^{k+1} - \psi_i^\top x^{k+1}), \ \forall i, \\ (\lambda_2)_i^{k+1} \leftarrow (\lambda_2)_i^k - \gamma_2\beta_2(\mathbf{y}_i^{k+1} - D_i x^{k+1}), \ \forall i, \\ (\lambda_3)^{k+1} \leftarrow (\lambda_3)^k - \gamma_3\beta_3(u^{k+1} - Cx^{k+1}). \end{cases} \tag{2.5}$$

In the ALM framework, each iteration requires an accurate minimization of $\mathcal{L}_{\mathcal{A}}$ jointly with respect to $\mathbf{y}, z, u$ and x, which can be expensive. It is easy to see that $\mathcal{L}_{\mathcal{A}}$ is separable with respect to $\mathbf{y}, z$ and u for fixed λ and x. Therefore, the per-iteration cost of the ALM is relatively expensive since it can not utilize such structures. To take full advantage of this structure, the *alternating direction method of multipliers* [17, 14] (or ADM) is applied, and it has recently been successfully applied to various signal and image reconstruction applications [46].

2.4.1 Alternating direction method of multipliers

Each iteration of ADM decreases $\mathcal{L}_A\left(\mathbf{y}, z, u, x, \lambda^k\right)$ via one round of alternating minimization with respect to $(\mathbf{y}, z, u)$ and x each; i.e.,

$$\begin{cases} \left(\mathbf{y}^{k+1}, z^{k+1}, u^{k+1}\right) \leftarrow \arg\min \mathcal{L}_A\left(\mathbf{y}, z, u, x^k, \lambda^k\right), \\ x^{k+1} \leftarrow \arg\min \mathcal{L}_A\left(\mathbf{y}^{k+1}, z^{k+1}, u^{k+1}, x, \lambda^k\right), \end{cases} \tag{2.6}$$

then followed by multiplier updates the same as in (2.5). For fixed $\lambda = \lambda^k$ and $x = x^k$, since $\mathcal{L}_A\left(\mathbf{y}, z, u, x^k, \lambda^k\right)$ is separable with respect to $\mathbf{y}$, z, and u each, the joint minimization of $(\mathbf{y}, z, u)$ can be carried out in parallel. In particular, z^{k+1} can be determined by

$$z_i^{k+1} \leftarrow \arg\min_{z_i} \varphi_1\left(z_i, \psi_i^\top x^k, (\lambda_1)_i^k\right) \triangleq S_1\left(\psi_i^\top x^k + (\lambda_1)_i/\beta_1, 1/\beta_1\right), \ \forall i, \tag{2.7}$$

where $S_1(\cdot, 1/\beta_1)$ is define to be the one-dimensional shrinkage operator given by $(\max\{|\xi| - 1/\beta_1, 0\} \cdot \operatorname{sgn}(\xi))$, where $\operatorname{sgn}(\cdot)$ is the signum function, $\mathbf{y}^{k+1}$ is determined by

$$\mathbf{y}_i^{k+1} \leftarrow \arg\min_{\mathbf{y}_i} \varphi_2\left(\mathbf{y}_i, D_i x^k, (\lambda_2)_i^k\right) \triangleq S_2\left(D_i x^k + (\lambda_2)_i^k/\beta_2, 1/\beta_2\right), \ \forall i, \tag{2.8}$$

where $S_2(\cdot, 1/\beta_2)$ is known as the two-dimensional shrinkage operator (which can be easily extended to high-dimensional cases with weights) defined as $(\max\{\|\mathbf{s}\| - 1/\beta_2, 0\} \cdot \mathbf{s}/\|\mathbf{s}\|)$, where $0 \cdot (0/0) = 0$ is assumed, and the minimization with respect to u is attained by

$$u^{k+1} \leftarrow \left(I + (\mu/\beta_3) P^\top P\right) u = C x^k + (\lambda_3^k + \mu P^\top b)/\beta_3, \tag{2.9}$$

here I represents the identity matrix. Since P is a selection matrix, $P^\top P$ is diagonal and thus the solution to (2.9) can be easily obtained. The computations of $\mathbf{y}$, z and u are linear in the dimension of x.

In the second step of ADM scheme (2.6), the minimization with respect to x is a least squares problem with the normal equations

$$Mx^{k+1} = D^\top(\beta_2 y^{k+1} - \alpha\lambda_2^k) + \beta\Psi^\top(\beta_1 z^{k+1} - \lambda_1^k) + C^\top(\beta_3 u^{k+1} - \lambda_3^k), \qquad (2.10)$$

where $M = \alpha\beta_2 D^\top D + \beta\beta_1\Psi^\top\Psi + \beta_3 C^\top C$ with $D \in \mathbb{R}^{2n^2\times n^2}$ being the global finite difference operator, and y^{k+1} is a reordering of $\mathbf{y}_i^{k+1}$, $i = 1, 2, \ldots, n^2$. Under the periodic boundary conditions, $D^\top D$ is block-circulant. Since C is a block-circulant matrix and further noting that Ψ is orthonormal, the coefficient matrix M of (2.10) is diagonalizable by the two-dimensional discrete Fourier transform. Therefore, the solution of (2.10) involves two fast Fourier transforms (FFTs). Finally, the Lagrangian multipliers as described in (2.5) are updated. The entire algorithm for (2.1) is summarized below, which can be shown to converge for equally valued $\gamma_i \in (0, (\sqrt{5}+1)/2)$ [13].

As opposed to RecPC, RecPF does not need the introduction of variable u and thus saves one split. Therefore, instead of (2.4), the augmented Lagrangian derived for RecPF is

$$\mathcal{L}_\mathcal{A}(\mathbf{y}, z, u, x, \lambda) = \alpha\sum\nolimits_i \varphi_2(\mathbf{y}_i, D_i x, (\lambda_2)_i) + \beta\sum\nolimits_i \varphi_1\left(z_i, \psi_i^\top x, (\lambda_1)_i\right) \qquad (2.11)$$
$$+\frac{\mu}{2}\|PFx - b\|_2^2,$$

Similar to RecPC the x-subproblem is readily diagonalizable and solved by calling FFTs. Also RecPF's fidelity term $\frac{\mu}{2}\|PFx - b\|_2^2$ possess a nicer structure. Expanding

---

ADM: Input problem data P, C, Ψ, b and model parameters $\alpha, \beta \geq 0$ and $\mu > 0$. Given $\beta_1, \beta_2, \beta_3, \gamma_1, \gamma_2, \gamma_3 > 0$, initialize $x = x^0$, $\lambda = \lambda^0$ and set $k = 0$.

While "not converged", **Do**

1) Obtain z^{k+1}, $\mathbf{y}^{k+1}$, and u^{k+1} according to (2.7), (2.8), and (2.9), respectively;

2) Compute x^{k+1} by solving (2.10) using FFTs.

3) Update λ according to Lines 2–4 of (2.5)

---

from both RecPF and RecPC, I create two methods gRecPF and gRecPC which are implementations on the GPU, respectively. The theory for these new methods is exactly that of their predecessors and is not the focus of this thesis. Instead, this work compares the CPU and GPU implementations for the reconstruction of signals using RecPF and RecPC.

Chapter 3

Graphics Processing Unit

Initially developed for the fast rendering of graphics, GPUs could only be used through special graphics libraries [5]. Yet, today GPUs are widely used because they have a very large memory bandwidth and huge computational performance [10]. In this thesis, I extend two existing methods, RecPF and RecPC, by using graphics hardware in order to achieve faster implementations. Also a closer look at a few key features of the GPU are provided in this chapter.

Graphics processing units have definitely moved away from only rendering graphics. Currently GPUs are capable of delivering cost-effective and energy-efficient performance in an array of applications such as a molecular dynamics [39], computational finance [20], and medical imaging [31] to name a few. However, the creation of efficient implementations for numerical algorithms on the GPU remains a daunting task due to the complexity of the architecture and technical specifications [5]. In order to get over this hurdle, I propose the use of a software platform called Jacket, created by AccelerEyes, which will be addressed in this chapter. Jacket is able to accelerate MATLAB code through the creation of optimized kernels (functions for the GPU) written in CUDA.

3.1 What is CUDA?

CUDA (or Compute Unified Device Architecture), released in 2007, is in fact two separate things. The first is a parallel computing architecture which moves away from the special graphics libraries used for interaction with the GPU. The second is a set of instructions used for the implementation of algorithms which is an extension of the C language. Thus the reader should note that the relevant CUDA definition will be based on context.

The popularity of CUDA can be somewhat credited to several strategic decisions from NVIDIA that have aided the popularity of the GPU [21]. According to Jen-Hsun Huang, Co-founder, President and Chief Executive Officer of NVIDIA, the GPU does not aim at replacing the central processing unit (CPU) but instead to work as a co-processor, assuming a more prominent role in personal computer system architecture [16]. NVIDIA has also been pushing to make every GPU CUDA–capable. The latter has allowed for NVIDIA to ride on the waves of the GeForce, which has consequently provided billions of dollars for research and development. Another outcome has been, that this decision has extended the reach of NVIDIA to hundreds of millions of personal computer users a year. These strategic decisions have helped propel and enhance general-purpose computing on the graphical processing unit (GPGPU).

3.2 CUDA as a language

CUDA serves as an interface between the CPU and the memory associated with it, which is referred to as the 'host', and the GPU, with its memory will be defined

to as the 'device'. The reader can recall these definitions easily by remembering that the GPU is hosted by the CPU. In order to start computation on the GPU the programmer first needs to allocate and set memory appropriately, because the CPU and GPU have separate memory pools. This is done through the use of two functions namely `cudaMalloc` and `cudaMemcpy` [10]. The first takes care of the allocation process while the second function transports the information from the host to the device.

Once information has been moved to the GPU the developer is able to use familiar tools to create functions called kernels that use parallel computation elements, known as threads. A simple overview of the process needed for GPGPU is given in Figure 3.1. The procedure begins with the allocation of data both on the host and device, followed by copying of information from the host to the device. Then the GPU is instructed to perform computation and finally the result is copied back to the host from the device. If the user has poor management of memory, then the results could be artificially bottlenecked through serialized data access [5]. Also memory transfers have the potential of becoming a bottleneck for GPU computation, so programmers must keep in mind how to best access and copy information [42]. There exist more technical attributes about CUDA which cannot be covered in full detail, but more information can be found in [10].

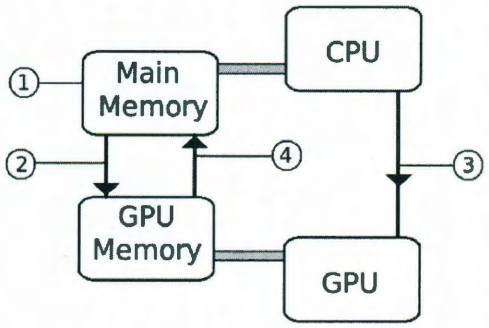


Figure 3.1 : CUDA processing flow. 1. The allocation of memory both on the host and the device. 2. A copy data from main memory to GPU memory. 3. CPU instructs the GPU to begin processing. 4. The result is copied from GPU to the main memory. Adapted from [10].

3.3 CUDA on the GPU

A great advantage of CUDA is the highly parallel nature of the architecture which has allowed the newer generation cards to redefine high performance computing [10]. This accounts for the capability of executing billions of calculations per second and is thus responsible for the surge of attention on GPU computing.

Even with the possibility of tremendous computational gain, there do exist some drawbacks to using the GPUs. For example, the widely advertised speed ups and peak performance rates are usually not easily achieved. However, CUDA does allow the programmer a lot of flexibility when implementing code, yet there exist some limiting factors. The first limiting factor deals with the performance rates where in order to reach the largest Gigaflop counts, close to peak performance, computations should be carried out in single precision. The gap between the single and double precision computations may differ greatly depending on the GPU card being used.

For example, single precision rates are about 12 times faster than those for double precision on the Tesla c1060 in theory. Yet, the Tesla c1060 is an older generation card. Newer generation cards such as the Tesla c2050 have a computational capability of over 1000 Gigaflops for single precision and over 500 Gigaflops for double precision, which has reduced the performance gap based on accuracy [33]. Also, implementations containing branching play a part in reducing the computational gain.

Nevertheless when used correctly, the GPU is able to provide great performance and all in a fraction of the time compared to its CPU counterparts. For the compressive sensing methods and applications, in this thesis the issue on using single or double precision becomes irrelevant as single precision is sufficient. Independent of NVIDIA there exist several third party wrappers for languages such as Python [23], Perl [34], Java [45], and MATLAB [1], to name a few, which allow users to interact with the GPU in languages they are more comfortable with.

3.4 Jacket

In general, programming on GPUs for scientific applications remains a difficult task due to the requirement that the user assimilate to new programming paradigms and sometimes even application programming interfaces (API). Jacket is a software platform that allows for the rapid development of GPGPU applications within the MATLAB computing environment as it transparently compiles and executes CUDA code on the GPU [2]. Thus Jacket provides an interface that is used in conjuction with MATLAB to interact with CUDA code which is then executed on the GPU.

This arrangement allows for users to interact only with the M-language creating single threaded M-codes and transforming them to GPU-enabled applications which utilize the low-level GPGPU graphics and computing capabilities [1]. The great advantage of Jacket is that minimal knowledge and time requirement are needed for implementations making it very attractive to MATLAB programmers that wish to use the GPU in computation.

3.5 Jacket by example

The processing flow that was described in Figure 3.1 does not change when using Jacket. Here I present how to handle the allocation and transfer of data followed by computation. For example, in MATLAB in order to create a single precision matrix A, the function `single`(·) is applied to the matrix, that is `single`(A). This creates the matrix A on the the host with single precision. In order to create the same type of matrix on the GPU a similar function `gsingle`(·) is used. This function allocates space for the matrix on the device and then transfers the single precision matrix to the GPU. The function `gdouble`(·) takes care of the double precision case.

Moreover, because Jacket inherently allows the same implementation in the MATLAB environment to hold for both the host and device data types, overloading of functions and operators is required. What is meant by function overloading is simply, the ability to use methods with the same name but which differ from each other in the number and type of terms they use for input and output. The addition ('+') function is a typical example, as seen in Figure 3.2.

MATLAB		with Jacket	
`A = rand(4);`	`B = rand(4);`	`X = rand(4);`	`Y = rand(4);`
`A = single(A);`	`B = single(B);`	`X = gsingle(X);`	`Y = gsingle(Y);`
`C = A + B;`		`Z = X + Y;`	

Figure 3.2 : Function overloading for Jacket. Random single precision matrices in $\mathbb{R}^{4\times4}$ are created. A, B are on the host and X, Y are on the device. The gsingle() function handles the allocation and transfer of data to the device. The '+' operator is overloaded to handle both types with seamless interaction for the programmer.

In this example (figure 3.2), for the MATLAB implementation the random matrices A, B $\in$ $\mathbb{R}^{4\times4}$ are first created with random entries (on the host) and set to have single precision. Then the addition of two matrices by use of '+' performs as expected and the value is stored into the matrix C, which inherits the type of A and B as a result.

In a similar fashion the matrices X, Y $\in$ $\mathbb{R}^{4\times4}$ are created and filled. At this point the matrices reside on the host machine. It is not until the call of the `gsingle` function that the resulting matrices X and Y get transferred to the device, with single precision. The addition of these two matrices results in the implicit creation of the matrix Z, which inherits the types of the input, namely a single precision matrix on the device.

Following the CUDA processing flow I developed gRecPF and gRecPC for the GPU using Jacket. The first step is to allocate memory on the device, which is trivial in Jacket as was seen in the last section. Although in order to have access to the classes provided by Jacket the user must ensure that a path to the software is provided. An example used in my implementations can be seen in Figure 3.3.

```
path_to_jacket='/usr/local/jacket';
addpath([path_to_jacket '/engine']);
addpath([path_to_jacket '/gfx']);
addpath([path_to_jacket '/gfx/mgl']);
```

Figure 3.3 : This is the path to Jacket libraries that needs to be place at the top of your M-language program to access Jacket functions.

This path gives the user access to the MATLAB functions that have been overloaded, as well as the functions created by Jacket for interaction with the GPU. The second step in the process calls for a copy of data from the host to the device, but Jacket takes care of updating information to the device as necessary. The third step, which involves the GPU being instructed to begin processing, is also taken care of by Jacket. Although, Jacket removes the burden of transfer and execution from the user, it cannot handle the copying of results from the device to the host. The user has to complete the transfer of information by casting variables to some host data type.

```
A = rand(4, gsingle);
B = single(A);
```

Figure 3.4 : The random single precision matrix A is created with the overloaded rand function onto the device. The values of the matrix are transferred and stored in B (on the host) by casting to a host data type.

3.6 Drawbacks to using Jacket

The key reason for using Jacket is to try and capture greater computational performance while not having to become an expert CUDA programmer. However, there are many factors which contribute to the speedups achieved with Jacket. The most obvious factor affecting the speedup is based on which one of NVIDIA graphics card is being used. The more advanced the card, the greater the speedup one can achieve [2]. Also the size of the problem and amount of data play a role as GPUs can outperform CPUs to a larger degree when data sizes increase. This goes back to the issue of data starvation, since GPUs exploit data parallelism large amounts of data is needed to make them faster than CPUs. Also the specific applications and implementations being looked at could influence speedup. Further information on how Jacket was was used to produce reconstruction is less time is provided in the next chapter.

Chapter 4

Methods

The intent of this thesis is to create faster reconstructions of signals in an efficient manner, that is with less information and faster recovery time. To this effect this thesis will show that compressive sensing permits the reconstruction of sparse compressible signals using less information. Also in order to speedup the reconstruction the use of the GPU is proposed. In particular, in order to create simple and practical implementations of existing CS methods on the GPU, Jacket will be used.

In this chapter more details about Jacket and issue related to the implementations are discussed. Jacket provides a collection of overloaded GPU functions (a full list can be found in [3]), however Jacket does not provide a complete set. This should not be seen as a problem, since Jacket is a wrapper and allows for a straightforward creation of new functions.

4.1 Implementation using Jacket

Building from the existing source code implementations of the RecPF [47] and the RecPC [49] methods, I was able to create Jacket implementations that provided faster reconstructions. The very first thing the reader will observe when comparing both the CPU and the GPU implementations for the methods is the length of the

code. The reason for this is not that Jacket allows for code to be carried out in a more concise fashion, but really it is as a consequence of careful consideration during implementation.

4.2 Minimizing branching

Recall that branching plays a role in the potential speedup of a method on the GPU. Since it could potentially prevent Jacket from creating optimal kernels, many of the unnecessary `if` statements in both methods have been removed. In creating the GPU implementations I made sure to reduce the number of branching (conditional) statements used, leaving only the essential branching statements in the algorithms. For example, neither implementation, i.e. gRecPF nor gRecPC, have the normalization portions of the code which are found in their CPU counterpart, because this would render three separate branches that the GPU kernels would have to test, even when the first `if` statement is false (see Figure 4.1).

4.3 Using fewer FFTs

The fast Fourier transform (FFT), an important and widely used numerical method in many science and engineering fields, plays a crucial role in my reconstructions [9]. Since the application at hand is the reconstruction of MR images, which contain samples that are in the spectral domain my code tries to avoid the over use of fast Fourier transforms (FFTs). Overuse of the FFT function, no matter how fast, with considerable iterations will delay the reconstruction.

```
%% normalize parameters and data
if opts.normalize
   if (~isreal(URange)||~isscalar(URange)||URange<eps)
      error('URange must be postive and real.');
   end

   fctr = 1/URange;
   B = fctr*B;

   if exist('uOrg','var');
      uOrg = fctr*uOrg;
      snr(U,uOrg);
   end

   aTV = nnz(picks)/sqrt(m*n)*aTV;
   aL1 = nnz(picks)/sqrt(m*n)*aL1;
end
```

Figure 4.1 : The implementation of this code would lead to three separate branches. In order to avoid any delays from excess branching, my implementations in Jacket allows a minimal number of conditional statements to be used.

A serious bottleneck occurs while performing the transforms on the GPU as the signal is required to be moved to and from the graphics card [30]. In an effort to keep the number of memory transfers down, the variable u in the code is kept in the spectral domain, that is I work with $\mathcal{F}(u)$ throughout all of the iterations. This allows for a reduction in the number transforms needed. Whenever the tolerance is met, the signal is converted back to the spatial domain. This simple step allows for a saving in computation for both the $\mathbf{y}$ and z subproblems.

4.4 Finite difference using FFTs

Because not all of the MATLAB functions used in the original CPU implementations existed for GPU data types in Jacket, I found that I needed to improvise when computing finite differences. I discovered that I could no longer use the `psf2otf` (point-spread function to optical transfer function). Instead I used an FFT to update the gradient information. The implementation of this approach can be seen in Figure 4.2, where this is for the finite difference along the x direction.

```
Ux = gzeros(m,n);

otf_DxtU = gsingle(psf2otf([0, -1, 1],[m,n]));

DxtU = otf_DxtU.*fft2(Wx-bx);
```

Figure 4.2 : The gradient information is updated using and FFT, making this more cost efficient than the CPU counterpart.

The GPU implementations created have been optimized in regards to issues that would prolong reconstruction times such as memory transfers, branching and operation selection. The complete implementations of gRecPF and gRecPC can be seen in appendices A and B, respectively. The next chapter presents numerical results.

Chapter 5

Numerical Simulations and Results

The results from the Jacket implementation of gRecPF and gRecPC are presented in this chapter along with a discussion of the numerical experiments. The main points of comparison are between the run times for the MATLAB implementations running on the CPU of the host machine (which are RecPF and RecPC), and the run times for the GPU implementations created with Jacket (gRecPF and gRecPC respectively).

The CPU and GPU implementations use the same MATLAB code for the problem set up and the presentation of results. The key difference in all of the methods is the solvers. The GPU implementations have function names beginning with a 'g'. The optimized CPU solvers from [47] and [49] were used in the CPU implementations.

5.1 Numerical Simulations

The computer used in executing all of the methods is a Lenovo D20 Workstation which is equipped with 2 Intel Xeon Processor E5506. These chips are quad core processors with a clock speed of 2.13 GHz each with 4 MB Cache. This workstation is also equipped with 10 GB of DDR3 RAM at 1066 MHz and an NVIDIA Tesla c1060 GPU. This Tesla c1060 contains 240 processor cores, with a clock speed of 1296 MHz each all tied together with a 512-bit bus. The methods were implemented

using Jacket v1.2.2 (build 3170) on MATLAB 7.9.0.529 (R2009b) with CUDA v2.3 used for compiling.

Both the GPU and CPU, computations have been carried out in single precision. The Tesla c1060 cards do support double-precision arithmetic, but has a substantially lower peak performance. Also, unlike the work by Lee and Wright, neither RecPF nor RecPC has trouble converging to the solution due to lack of accuracy [24].

An issue regarding the initialization of GPU computations was encountered while running numerical experiments. The very first call to the GPU functions takes significantly longer than subsequent calls, in some instances nearly 1 second longer. To avoid this issue a "warm up" run was executed to get communication started. The "warm up" process is all right since real situations will make multiple calls to the solver. An example, could be solving different time slices in a signal sequence, or possibly to be used as one of a number of tasks.

5.2 Results

The quality in the solutions both for the GPU and CPU implementations is similar. Plots showing the total run time for the solvers after a few warm-up computations are presented.

5.2.1 RecPF versus gRecPF

For the reconstruction of signals using these methods I generated my test set using the Shepp-Logan phantom image, with sampling ratio of about 20 percent. For this

test, the generated data was retrieved by $f_p = P\mathcal{F}x$, but first rescaling the intensity values of the tested image to [0, 1] followed by applying a partial FFT and no noise was added. In order to apply a partial FFT, a number of radial lines (RLs) spread out from the center were used to sample the Fourier domain; for example, Figure 5.1 shows 22 radial lines in a Fourier domain.

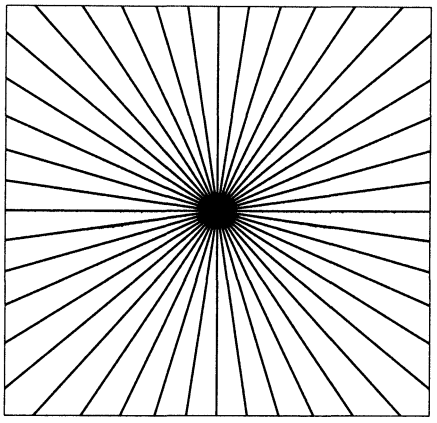


Figure 5.1 : This figure shows 22 radial lines used to sample in a Fourier domain.

In Figure 5.2 the times taken to reconstruct the phantom image are depicted where the host machine is using all 8 cores. From the figure it can be seen that the GPU implementation is about 3 times as fast as that of the CPU.

For Figure 5.3 a similar test was done except that the host machine was restricted to only using 1 of the 8 cores available. However, even with only one core being used the speedup for the GPU code is just slightly over three times.

Taking into consideration the overhead time for the GPU to start, the following test demonstrates 25 trials, where each trail takes 50 separate reconstruction times

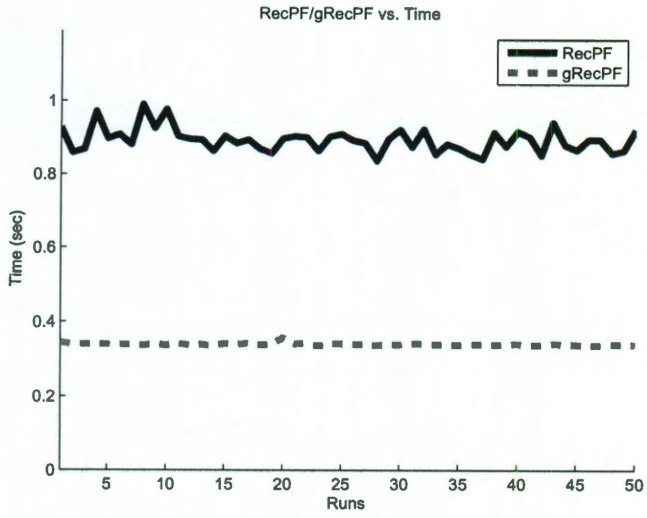


Figure 5.2 : This plot shows the time for reconstruction on the phantom (256) using all 8 cores of the workstation.

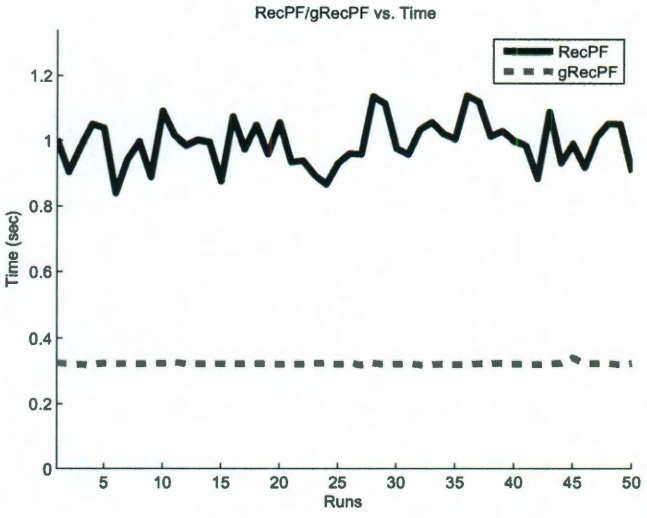


Figure 5.3 : This plot shows the time for reconstruction on the phantom(256) using only 1 core of the workstation.

and takes their arithmetic mean using a single core.

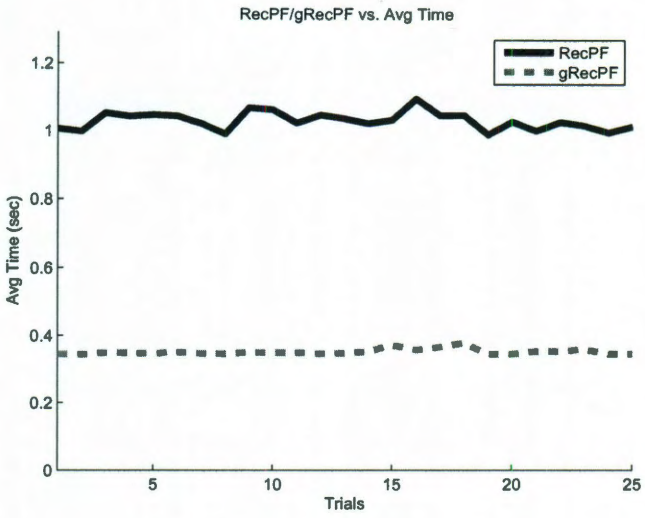


Figure 5.4 : This plot shows the average time for reconstruction on the phantom(256) using a single core. Average times here are calculated by taking the arithmetic average of 50 runs for each trial.

Because a "warm" up phase is required for single run reconstructions, here (Figure 5.5) I look at average runs but with a clean work space for every reconstruction. I still observe a speedup of about 3 times, however the times for the CPU reconstructions decreases compared to the other test. The trials here are the same as the previous plot.

5.2.2 RecPC versus gRecPC

For the reconstruction of signals using RecPC and gRecPC I generated the test set by using the Shepp-Logan phantom image, with sampling ratio of 15 percent. For this test, the generated data was retrieved by taking random complex subsamples of

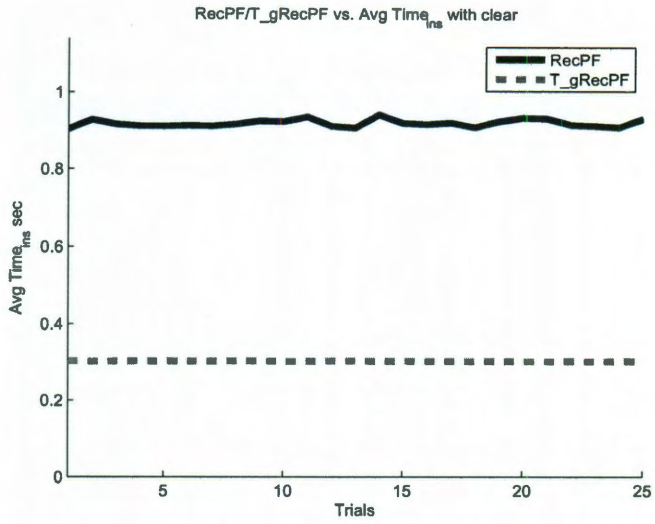


Figure 5.5 : Average time for reconstruction of the phantom(256) using only 1 core when the workspace was cleared after each 'run'. Trials are defined similarly to previous figure. .

a Fourier domain. For these test the size of the phantom image is increased by powers of 2. Also all figures are created using only a single core on the host machine as it was observed that no significant speedup was achieved from having all eight working.

In the first test a phantom image of size 256×256 was reconstructed 50 separate times, then the mean of the times was calculated was recorded as a trial. In Figure 5.6 there is a gain of above 2 times for the GPU implementation for all of the trails. Also it should be noted that the times for reconstruction for these methods are slower than those of RecPF and gRecPF, but the reason we are looking at this methods is that they provide an algorithm that has feasible hardware realization. If implemented onto hardware this reconstruction paradigm would not need to be run on a separate workstation, but could be implemented on the MRI scanner itself.

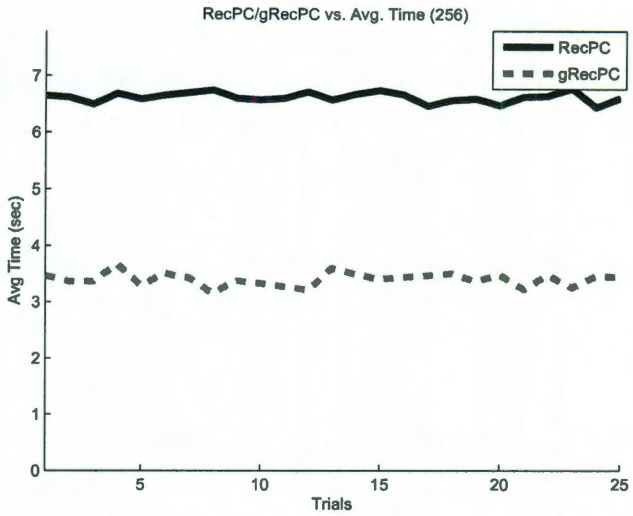


Figure 5.6 : This plot shows the average time for reconstruction on the phantom(256) where a speed up of about 2.5 times is seen over the CPU implementation.

Figure 5.7 a speed up of about 5 times is seen across all of the trials. This shows the potential that the GPU has over the CPU as more information becomes available there is more payoff for using graphics hardware as more computations can be done.

In Figure 5.8 the size of the phantom image is increased from a 16×16 to an image of 2048×2048. The image is doubled every time and the best of two times for reconstructing is recorded. From the figure it is clear that the over head for using the GPU is greater than the advantage for images below 128×128. After this point the speedup for reconstruction almost grows linearly with the size of the image.

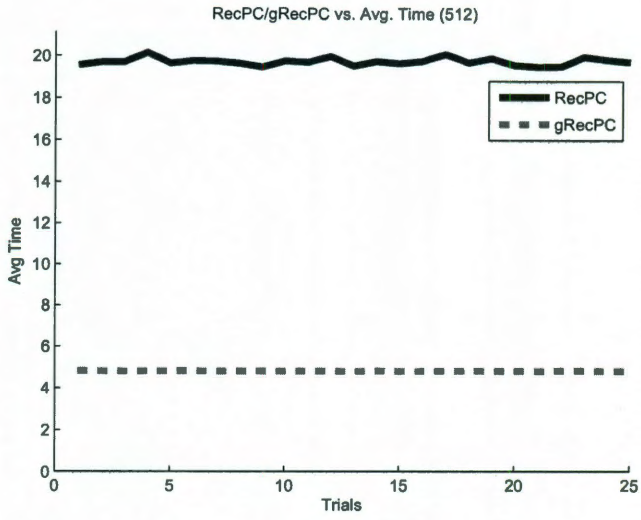


Figure 5.7 : This plot shows the average time for reconstruction on the phantom(512) where about a 5 times speed up is observed.

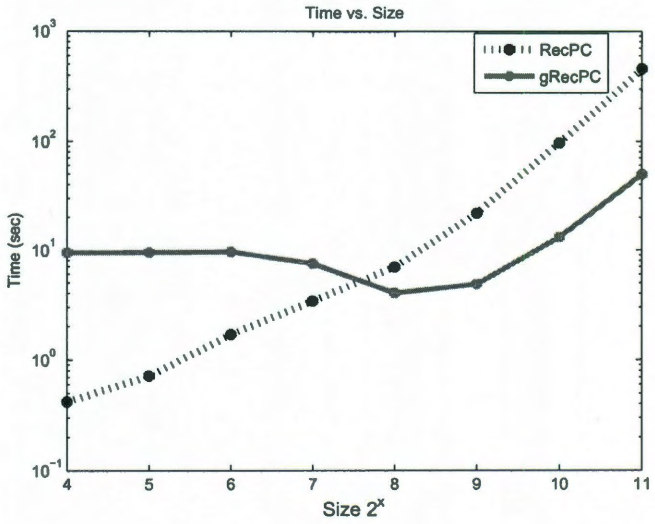


Figure 5.8 : This figure demonstrates the time for reconstruction on phantom images ranging in size from 16-2048. It can also be gathered from this image that the size of the signal would have to be larger than 256 in order for the gRecPC to beat RecPC.

Chapter 6

Conclusions

Traditional reconstruction of signals has provided much guidance throughout the years, but as the amount of information keeps growing it is clear that this sensing protocol maybe inefficient due to under utilization of data. Also, the sample rate of the Nyquist sampling theorem becomes too expensive if not impossible for some signal acquisitions systems [4]. In order to address this, compressive sensing techniques and algorithms have been utilized in this thesis to reduce the number of samples required to reconstruct signals.

However, changing the signal acquisition protocol is not enough. The advantage of compressive sensing is that it is not wasteful when it comes to sampling, but there is a price to be paid and that comes when reconstructing the signal. Because of the small number of measurements, the problem at hand becomes an under determined linear system requiring some computational performance to solve. As the optimization problem becomes more complicated, the computational demand is increased. This is where the use of the GPU is best suited. The creation of methods for the GPU provide the computational performance required to efficiently solve large inverse problems. In this thesis, it has been demonstrated that practical CS implementations tailored for the GPU provide a route for using less information while giving reconstructions faster.

An area of study, to which this work would be of particular interest is medical

imaging systems. The ability to solve signal reconstructions faster would promote a faster sampling time for example in an MRI session. This would have positive benefits considering patients would not have to be restrained for long periods of time. Another benefit would be that more patients could be seen in the same amount of time. Thus clinics could potentially reduce the cost of an MRI session as their return on investment would be higher.

This work is one of the few works that provides ideas from two very popular fields CS and GPGPU. It provides fast and cost efficient methods for solving signal reconstruction problems and shows some room for improvement.

6.1 Future Work

A possible direction for this work to continue would involve implementations that use the CUDA programming language. It is expected that the C language extension can provide greater flexibility in terms of the operations chosen in the implementation. Also a more in depth look at the terms used in the optimization problems of RecPF and RecPC could help reduce the number of computations required. For example, the substitution of a median formula term as a replacement for the total variation [25].

Appendix A

Jacket implementation of gRecPF

```
function [U,Out]=gRecPF(m,n,aTV,aL1,picks,B,opts,URange,uOrg)

   %% set options  --- setting from opts ---
   maxItr     = opts.maxItr;
   gamma      = opts.gamma;
   beta       = opts.beta;
   relchg_tol = opts.relchg_tol;

   bPrint = false;   % turning on slows code down

   % ------------------------------------------------------------
   %% initialize variables

   U = gzeros(m,n);

   snr(U,uOrg);   % initial signal to noise ratio

   % ------------------------------------------------------------
   %% initialize constant parts of numerator and denominator
   %  (in order to save on computation time)

   Numer1 = zeros(m,n);
   Numer1(picks) = sqrt(m*n)*B;
   Numer1 = gsingle(Numer1);

   Denom1 = zeros(m,n); Denom1(picks) = 1;
   Denom1 = gsingle(Denom1);

   prd = sqrt(aTV*beta);

   tmp        = zeros(m,n);
   tmp(1,1)   =  prd;
   tmp(1,end) = -prd;
```

```
first = fft2(tmp);

tmp(1,end) =  0;
tmp(end,1) = -prd;

second = fft2(tmp);

Denom2 = gsingle( abs(first).^2 + abs(second).^2 );

Denom = Denom1 + Denom2;

% ------------------------------------------------------------
%% initialize constants

Ux = gzeros(m,n);    Uy = gzeros(m,n);
bx = gzeros(m,n);    by = gzeros(m,n);

otf_DxtU = gsingle(psf2otf([0, -1, 1],[m,n]));
otf_DytU = gsingle(psf2otf([0; -1; 1],[m,n]));

otf_x = gsingle(psf2otf([1, -1],[m,n]));
otf_y = gsingle(psf2otf([1; -1],[m,n]));

tau         = gsingle(1/beta);
aTV         = gsingle(aTV);
relchg_tol = gsingle(relchg_tol);

% ------------------------------------------------------------
%% Main loop

for ii = gsingle(1:maxItr)

   % ================================
   %  Begin Alternating Minimization
   % ----------------
   %   W-subproblem
   % ----------------
   % isotropic TV

   UUx = Ux + bx;       UUy = Uy + by;

   V = sqrt(UUx.*conj(UUx) + UUy.*conj(UUy));
```

```
V = max(V - tau, 0) ./ max(V,eps);

Wx = V.*UUx;       Wy = V.*UUy;

% ----------------
%   U-subproblem
% ----------------

Uprev = U;

DxtU = otf_DxtU.*fft2(Wx-bx);
DytU = otf_DytU.*fft2(Wy-by);
rhs  = (aTV*beta)*(DxtU + DytU);

U = (Numer1 + rhs)./Denom;

% ------------------------------------------------------
% Update quantities of U

Ux = ifft2(otf_x.*U);
Uy = ifft2(otf_y.*U);

%
%  End Alternating Minimization
% ================================

% ------------------------------------------------------------
% check stopping criterion
%

relchg = norm(U - Uprev,'fro')/norm(U,'fro');

if bPrint;
   fprintf('itr=%3d relchg=%4.1e', single(ii), single(relchg));

   if exist('uOrg','var');
      fprintf(' snr=%4.1f',snr(real(ifft2(U))));
   end

   fprintf('\n');
end
```

```
      if (relchg < relchg_tol); break; end

      % ------------------------------------------------------------
      % ADM update
      %

      bx = bx + gamma*(Ux - Wx);
      by = by + gamma*(Uy - Wy);

   end % outer

   U = ifft2( U );

   Out.iter = ii;

   % -------------------- cast to real? ------------------------
   if opts.real_sol; U = real(U); end
end
```

Appendix B

Jacket implementation of gRecPC

```
function [U,Out]=gRecPC(m,n,aTV,aL1,picks,Mask,B,opts,URange,uOrg)

   %% set options  --- setting from opts ---

   maxItr     = opts.maxItr;
   gamma      = opts.gamma;
   beta1      = opts.beta1;
   beta2      = opts.beta2;
   beta3      = opts.beta3;
   bSymm      = opts.bsymm;
   relchg_tol = opts.relchg_tol;

   bPrint = false;    % turning on slows code down

   % ------------------------------------------------------------
   %% initialize variables

   U     = gzeros(m,n));
   g     = gzeros(m,n));
   CircU = gzeros(m,n));

   snr(U,uOrg);   % initial signal to noise ratio

   % ------------------------------------------------------------
   %% initialize constant parts of numerator and denominator
   %  (in order to save on computation time)

   Numer1 = zeros(m,n);
   Numer1(picks) = B/beta3;
   Numer1 = gsingle(Numer1);

   D      = gsingle(psf2otf(Mask));
   Ds     = conj(D);
```

```
Denom1 = beta3*real(D.*Ds);
Denom1 = gsingle(Denom1);

prd = sqrt(aTV*beta1);

tmp         = zeros(m,n);
tmp(1,1)    =  prd;
tmp(1,end) = -prd;

first = fft2(tmp);

tmp(1,end) =  0;
tmp(end,1) = -prd;

second = fft2(tmp);

Denom2 = gsingle(abs(first).^2 + abs(second).^2);

Denom = Denom1 + Denom2;

% ------------------------------------------------------------
%% initialize constants

Ux = gsingle(zeros(m,n));    Uy = gsingle(zeros(m,n));
bx = gsingle(zeros(m,n));    by = gsingle(zeros(m,n));

otf_DxtU = gsingle(psf2otf([0, -1, 1],[m,n]));
otf_DytU = gsingle(psf2otf([0; -1; 1],[m,n]));

otf_x = gsingle(psf2otf([1, -1],[m,n]));
otf_y = gsingle(psf2otf([1; -1],[m,n]));

tau         = gsingle(1/beta1);
aTV         = gsingle(aTV);
relchg_tol = gsingle(relchg_tol);

% ------------------------------------------------------------
%% Main loop

for ii = gsingle(1:maxItr)

   % ================================
```

```
%  Begin Alternating Minimization
% ----------------
%   W-subproblem
% ----------------
% isotropic TV

UUx = Ux + bx;       UUy = Uy + by;

V = sqrt(UUx.*conj(UUx) + UUy.*conj(UUy));
V = max(V - tau, 0) ./ max(V,eps);

Wx = V.*UUx;       Wy = V.*UUy;

% ----------------
%   V-subproblem
% ----------------

V = Numer1 + CircU - g;
V(find(picks)) = V(find(picks)) / (1+1/beta3);

% ----------------
%   U-subproblem
% ----------------

Uprev = U;

rhs1 = Ds.*fft2(beta3*(V+g));

DxtU = otf_DxtU.*fft2(Wx-bx);
DytU = otf_DytU.*fft2(Wy-by);
rhs  = (aTV*beta1)*(DxtU + DytU);

U = ( rhs1 + rhs )./Denom;     % intermediate U

CircU = ifft2(D.*U);

% ---------------------------------------------------------
% Update quantities of U

Ux = ifft2(otf_x.*U);
Uy = ifft2(otf_y.*U);
```

```
      %
      %  End Alternating Minimization
      % ================================

      % ------------------------------------------------------
      % check stopping criterion
      %

      relchg = norm(U-Uprev,'fro')/norm(U,'fro');

      if bPrint;

         fprintf('itr=%3d relchg=%4.1e', single(ii), single(relchg));
         fprintf(' diff_CU=%4.1e', single(norm(CircU-V,'fro')));

         if exist('uOrg','var');
            fprintf(' snr=%4.1f',single(snr(real(ifft2(U)))));
         end

         fprintf(' ||PV-B||=%4.1e\n', single(norm(V(find(picks))-B)));
      end

      if (relchg < relchg_tol); break; end

      % ------------------------------------------------------------
      % ADM update
      %

      bx = bx + gamma*(Ux - Wx);
      by = by + gamma*(Uy - Wy);
      g  = g  + gamma*(V - CircU);

   end % outer

   U = ifft2(U);

   Out.iter = ii;

   % -------------------- cast to real? -----------------------
   if opts.real_sol; U = real(U); end
end
```

Bibliography

[1] AccelerEyes, 75 5th Street NW, Suite 204, Atlanta, GA 30308. *Jacket: GPU Computing without CUDA programming.* www.accelereyes.com.

[2] AccelerEyes. Jacket Documentation, November 2010. http://wiki.accelereyes.com/wiki/index.php/Jacket_Documentation.

[3] AccelerEyes. Function List, March 2011. http://wiki.accelereyes.com/wiki/index.php/Jacket_Function_List.

[4] R. G. Baraniuk. Compressive Sensing [Lecture Notes]. *Signal Processing Magazine, IEEE*, 24(4):118–121, July 2007.

[5] A. Borghi, J. Darbon, S. Peyronnet, T. Chan, and S. Osher. A Compressive Sensing Algorithm for Many-Core Architectures. In G. Bebis, et al., editor, *Advances in Visual Computing*, volume 6454 of *Lecture Notes in Computer Science*, pages 678–686. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-17274-8_66.

[6] E.J. Candes. Compressive sampling. In *Proceedings of the International Congress of Mathematicians*, Madrid, Spain, 2006.

[7] E.J. Candes and M.B. Wakin. An Introduction To Compressive Sampling. *Signal Processing Magazine, IEEE*, 25(2):21–30, March 2008.

[8] Emmanuel J. Candes, Justin K. Romberg, and Terence Tao. Stable Signal Recovery from Incomplete and Inaccurate Measurements. *Communications on Pure and Applied Mathematics*, 59(8):12071223, August 2006.

[9] NVIDIA Coporation. *CUDA CUFFT Library.* Version 1.2, October 2007.

[10] NVIDIA Coporation. *NVIDIA CUDA C Programming Guide.* Version 3.2., October 2010.

[11] R. Courant. Variational Methods for the Solution of Problems of Equilibrium and Vibrations. *Bull. Amer. Math. Soc.*, 49:1–23, 1943.

[12] D. Donoho. Scanning the Technology. *Proceedings of the IEEE*, 98(6):910–912, June 2010.

[13] M. Fortin and R. Glowinski. *Augmented Lagrangian Methods.* Elsevier Science Publishers, North-Holland, Ambsterdam, New York, 1983.

[14] D. Gabay and B. Mercier. A Dual Algorithm for the Solution of Nonlinear Variational Problems via Finite-Element Approximations. *Computers and Mathematics with Applications*, pages 17–40, 1976.

[15] D. Geman and Chengda Yang. Nonlinear Image Recovery with Half-Quadratic Regularization. *Image Processing, IEEE Transactions on*, 4(7):932 –946, July 1995.

[16] Peter N. Glaskowsky. NVIDIA's Fermi: The First Complete GPU Computing Architecture. *NVIDIA Coporation*, September 2009.

[17] R. Glowinski and A. Marrocco. Sur lapproximation par elements finis dordre un, et la resolution par penalisation-dualite dune classe de problemes de Dirichlet nonlineaires. *Rev. Francaise dAut. Inf. Rech. Oper.*, pages 41–76, 1975.

[18] L. He, T.-C. Chang, S. Osher, T. Fang, and P. Speier. MR Imange Reconstruction by using the Iterative refinement Method and Nonlinear Inverse Scale Space Methods. Technical report, UCLA, 2006. CAM Report 06-35.

[19] Magnus R. Hestenes. Multiplier and Gradient Methods. *Journal of Optimization Theory and Applications*, 4(5):303–320, November 1969.

[20] Jen-Hsun Huang. Jen-Hsun Huang Announcing Fermi. Recorded presentation, September 2009. http://www.nvidia.com/object/fermi_architecture.html.

[21] Jen-Hsun Huang. Opening Keynote with Jen-Hsun Huang,NVIDIA. Recorded presentation, September 2010. http://www.nvidia.com/object/gtc2010-presentation-archive.html.

[22] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. Technical Report 2009-40, Scientific Computing Group, Brown University, Providence, RI, USA, November 2009.

[23] A. Kloeckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA: GPU Run-Time Code Generation for High-Performance Computing. Technical Report 2009-40, Scientific Computing Group, Brown University, Providence, RI, USA, November 2009.

[24] Sangkyun Lee and Stephen Wright. Implementing Algorithms for Signal and Image Reconstruction on Graphical Processing Units. *http://dsp.rice.edu/cs*, 2008.

[25] Yingying Li and Stanley Osher. Coordinate Descent Optimization for ℓ^1 Minimization with Application to Compressed Sensing; a Greedy Algorithm. Technical Report 09-17, UCLA, March 2009.

[26] Dong Liang, Guangwu Xu, Haifeng Wang, K.F. King, Dan Xu, and Leslie Ying. Toeplitz Random Encoding MR Imaging using Compressed Sensing. In *Biomedical Imaging: From Nano to Macro, 2009. ISBI '09. IEEE International Symposium on*, pages 270 –273, July 2009.

[27] Michael Lustig, David Donoho, and John M. Pauly. Sparse MRI: The Application of Compressed Sensing for Rapid MR Imaging. *Magnetic Resonance in Medicine*, 58(6):1182–1195, 2007.

[28] S. Ma, W. Yin, Y. Zhang, and A. Chakraborty. An effcient algorithm for compressed mr imaging using total variation and wavelets. *IEEE International Conference on Computer Vision and Pattern Recognition (CVPR) 2008*, 2008.

[29] Jia Meng, Wotao Yin, Husheng Li, E. Houssain, and Zhu Han. Collaborative Spectrum Sensing from Sparse Observations using Matrix Completion for Cognitive Radio Networks. In *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, pages 3114–3117, March 2010.

[30] Kenneth Moreland and Edward Angel. The FFT on a GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, pages 112–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[31] Brian E. Nett, Jie Tang, and Guang-Hong Chen. GPU Implementation of Prior Image Constrained Compressed Sensing (PICCS). volume 7622. SPIE, 2010.

[32] NVIDIA Corporation. *What is GPU Computing?* http://www.nvidia.com/object/GPU_Computing.html.

[33] NVIDIA Corporation. *NVIDIA Tesla C2050 / C2070 Datasheet* , 2010. http://www.nvidia.com/object/tesla_product_literature.html.

[34] POGL: The Perl OpenGL. A Portable Perl Binding for OpenGL, 2007. http://graphcomp.com/pogl.cgi.

[35] M. J. D. Powell. A Method for Nonlinear Constraints in Minimization Problems. *Optimization*, pages 283–298, 1972.

[36] J. Romberg. Imaging via Compressive Sampling. *Signal Processing Magazine, IEEE*, 25(2):14–20, March 2008.

[37] Fadil Santosa and William W. Symes. Linear Inversion of Band-Limited Reflection Seismograms. *SIAM Journal on Scientific and Statistical Computing*, 7(4):1307–1330, 1986.

[38] Jianing Shi, Wotao Yin, Stanley Osher, and Paul Sajda. A Fast Hybrid Algorithm for Large-Scale l1-Regularized Logistic Regression. *Journal of Machine Learning Research*, 11:713–741, March 2010.

[39] John E. Stone, David J. Hardy, Ivan S. Ufimtsev, and Klaus Schulten. GPU-accelerated Molecular Modeling Coming of Age. *Journal of Molecular Graphics and Modelling*, 29(2):116 - 125, 2010.

[40] Yaakov Tsaig and David L. Donoho. Extensions of Compressed Sensing. *IEEE Trans. Inform. Theory*, 52:1289–1306, 2006.

[41] S. Vasanawala, M. Alley, R. Barth, B. Hargreaves, J. Pauly, and M. Lustig. Faster Pediatric MRI via Compressed Sensing. In *Proc. Annual Meeting Soc. Pediatric Radiology (SPR)*, Carlsbad, CA, April 2009.

[42] Wolfram Research Inc. *CUDALink User Guide.* http://reference.wolfram.com/mathematica/CUDALink/guide/CUDALink.html.

[43] J. Wright, A.Y. Yang, A. Ganesh, S.S. Sastry, and Yi Ma. Robust Face Recognition via Sparse Representation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 31(2):210–227, February 2009.

[44] S.J. Wright, R.D. Nowak, and M.A.T. Figueiredo. Sparse Reconstruction by Separable Approximation. *Signal Processing, IEEE Transactions on*, 57(7):2479–2493, July 2009.

[45] Yonghong Yan, Max Grossman, and Vivek Sarkar. JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 887–899, Berlin, Heidelberg, 2009. Springer-Verlag.

[46] Junfeng Yang and Yin Zhang. Alternating Direction Algorithms for ℓ_1-problems in Compressive Sensing, 2009. CAAM Technical Report TR09-37.

[47] Junfeng Yang, Yin Zhang, and Wotao Yin. *Source code for RecPF: Reconstruction from Partial Fourier data* . Rice University. http://www.caam.rice.edu/~optimization/L1/RecPF/.

[48] Junfeng Yang, Yin Zhang, and Wotao Yin. A Fast Alternating Direction Method for TVL1-L2 Signal Reconstruction From Partial Fourier Data. *Selected Topics in Signal Processing, IEEE Journal of*, 4(2):288–297, April 2010.

[49] W. Yin, S. Morgan, J. Yang, and Y. Zhang. *Source code for RecPC: Practical Compressive Sensing with Toeplitz and Circulant Matrices* . Rice University. http://www.caam.rice.edu/~optimization/L1/RecPC/.

[50] W. Yin, S. Morgan, J. Yang, and Y. Zhang. Practical Compressive Sensing with Toeplitz and Circulant Matrices. *Rice University CAAM Technical Report TR10-01*, 2010.